

Migration between HI-TECH C Compiler for PIC10/12/16 MCUs and PICC™ STD Compilers

Produced on: June 14, 2011

Australian Design Centre
45 Colebard Street West
Acacia Ridge QLD 4110
Australia

web: http://www.htsoft.com

0.1 Introduction

This document describes the differences between the HI-TECH C Compiler for PIC10/12/16 MCUs and the PICC STD compilers, specifically in terms of the new features that the Omniscient Code Generation (OCG) compiler technology offers, as well as the impact these have on writing or modifying source code that is to be used with the HI-TECH C compiler. For further information about the HI-TECH C compiler for PIC10/12/16 MCUs, please refer to the release notes and the user manual.

0.2 Operational Differences

The fundamental difference between the HI-TECH C Compiler for PIC10/12/16 MCUs and the STD compiler on which it is based is that a new code generator has been developed to read and process all C source modules in one pass.

The HI-TECH STD compilers follow the traditional compiler approach of invoking the code generator separately for each C source module being compiled. These steps are then followed by a link step which combines the relocatable object files produced by the code generator. No information exchange occurs from one execution of the code generator to another, and hence the only information the code generator can gain from code defined in other modules is via the usual C declarations. Such declarations provide information which is limited purely to C types, and convey no information regarding usage or internal operation of functions.

HI-TECH compilers based on Omniscient Code Generation technology, or OCG compilers, allow all parts of the C program to be analyzed and information to be extracted from the complete program. The code generator becomes omniscient in that it is not limited to a piecemeal program analysis, each with a scope of just one module. The information extracted from a program even extends to modules within library files that were derived from C source code. A new library format — the p-code library file — allows this to happen. This new method of compilation uses a true multiple module approach and is in stark contrast to having all source files "included" into one module, which does not allow the same level of variable and function scope.

0.3 Functional Differences

With this new technology comes the capacity for the code generator to operate without many of the additional non-standard C qualifiers and compiler options that would need to be supplied and maintained by the programmer. This means code is easier to write and is more portable. In addition, many functional features which have been impractical can now be realized. This section describes the additional features that have already been added to the OCG compiler to take advantage of the information obtained from the new code generation technology.

Pointer variables. The size and scope of each pointer variable is now determined from its usage within the program being compiled. Each C code assignment to a pointer variable is tracked by the code generator allowing

a table to be constructed of every object that each pointer can reference. From this information the best size and format of the address that the pointer holds can be determined.

Customized printf. Each call to `printf` (and related) functions is now monitored by the code generator. The format string for each call is examined, and those placeholders which are used are collated. This information is then used to determine which components of the complete `printf` routine will be required by this program. A generic `printf` routine is then customized based on this information during the build process resulting in `printf` code no larger than required. This process even operates, albeit to a lesser degree, when the format string is not a string literal — in this case the additional parameters passed to the `printf` function can be used as an indication of the functionality required.

Unused functions. Functions which are defined, but never called, now no longer contribute to the generated output. This has always been true for library routines which were never called, but this now applies to user-defined functions as well.

Unused return values. If the return value of a function is not used by every expression that calls this function, then the code associated with evaluating the return value for that function is now not generated.

Function duplication The OCG compiler uses the same non-reentrant model as does the STD compiler. However, if a function is called from both an interrupt function (or any functions called from an interrupt function) and main-line code, the generated code associated with the called function is now duplicated. One copy of the function will be called from main-line code; the other from the interrupt code. This allows the programmer to call functions with less concern as to whether the target MCU allows reentrant function calls, and without having to maintain more than one copy of a source function. Both C- and assembly-derived library routines are also duplicated in the library files. The code generator determines which is the appropriate function to call in the generated assembly code.

0.4 Functional Considerations

The following are points which must be considered when writing code for HI-TECH C Compiler for PIC10/12/16 MCUs, or if migrating code from the PICC STD compiler.

Intermediate file format The STD compiler used the relocatable object file (extension `.obj`) as the intermediate file format. Each C source file is passed to the code generator, then the assembler so that for each C source file there will be one relocatable object file. As the OCG compiler performs code generation on all the C source files that make up a program, this intermediate file format is no longer applicable. Instead, the intermediate file format used is the p-code format (extension `.p1`). This file can be generated with the `--PASS1` compiler driver option. This file is produced before the code generation phase of compilation. You cannot use the `-C` option with the OCG compiler unless you have specified all the C source files to be compiled, or you are only compiling assembly source files.

Single output files In addition to only one relocatable object file built, there is also only one assembler list file. This list file will contain the assembly code and corresponding C source code, but it will contain the assembly listing for entire program, including assembly for C-derived library modules. This makes it easier to examine the assembly listing. The basename of both the assembly listing and relocatable object file is derived from the name of the first file listed on the command line, or typically the project name if using HI-TIDE™.

Psect allocation In the PICC STD compiler, all variables, with the exception of absolute variables, are placed in a psect and then that psect is positioned in memory by the linker. The Omniscient code generator performs many of the tasks that were performed by the linker, in particular memory allocation of some global and `static` local variables. Once allocated, these variables are assigned an address and behave as absolute variables. Initialised variables, `auto` variables and `const`-qualified variables are not allocated addresses by the code generator and they will be placed into a psect and positioned by the linker in the usual way. Overlaying of non-active functions' auto/parameter blocks is still performed as with the STD compiler.

Placement of variables The use of `#pragma psect` directives to redirect data within named psects will not work as expected — it will redirect all psects within the program, not just the module in which the pragma is located. Variables to be positioned at absolute locations can be done so using the `@` *`address`* construct. Multiple variables that must be placed as a block are best defined as a structure and made absolute. Initialized `const` objects cannot presently be placed at absolute locations, however a new mechanism will be introduced in latter releases to allow this.

Functions can now be placed at a specific address using a construct similar to that used by variables. Function prototypes followed by an `@ address` construct will be placed at the address specified.

One requirement of the programmer writing code for the PICC STD compiler was to manage the bank allocation of `static` or global variables with non-standard qualifiers such as `bank1`, `bank2` or `bank3`. The default behavior of the OCG compiler is now to ignore these qualifiers as bank allocation is managed by the omniscient code generator, however if the `--addrqual` option is given to the compiler driver, these qualifiers can be accepted as a request or requirements as where they should be located. This means that when `--addrqual=request` is used, a variable defined with a `bankn` qualifier will first attempt to be positioned in its nominated bank. If this is not possible, the variable will then attempt to be located in another bank. This differs from the PICC STD compiler which would generate an error if the variable failed to find a space in its nominated bank, resulting in a build failure. The behavior of the STD compiler can be mimiced with the `--addrqual=require` option.

Pointer assignments The OCG compiler tracks assignments to pointer variables. It looks at the type of the object from which the address is derived to determine the size and scope of the pointer. Assigning a literal address to a pointer will defeat this tracking mechanism as it will not be able to identify any source object. Always take the address of an appropriately defined label when making assignments to pointers, for example define an absolute variable (using the `@ address` construct) to overlay with the memory area, and assign the address of this absolute variable to the pointer. Ensure that the absolute object is of an appropriate type. If the pointer will be incremented to access a range of address, make the absolute variable an array with

the desired size. The compiler may use the size of the objects referenced by a pointer to determine the appropriate size and format for the pointer's address. Assigning "magic numbers" to pointers — even if casting the value — is not to be recommended with any compiler.

Auto variable block basename The prefix assigned to the basename of the `auto` variable block has been changed from `?a` to `??`. This change was necessary due to the way the symbol was used. Any assembly code that references this symbol will need to be adjusted.

Runtime startup code The OCG compiler automatically generates runtime-startup code in the same way as the STD compiler. However the code generated will differ in some ways to that generated by the STD compiler. If user-defined runtime startup code is being used, this may need to be updated.

New library files The OCG compiler uses different library files to those used by the STD compiler. In addition, all library files are p-code libraries (`.lpp` files). Any user-defined libraries generated from C source code will need to be regenerated. Both the linker and librarian applications shipped with HI-TECH C Compiler for PIC10/12/16 MCUs have been updated to handle both relocatable and p-code object files and libraries.

Temporary variables The PICC STD compiler used special temporary variables to store intermediate values and return values from C functions. These variables were contained in a special psect, called `temp`, and were referenced as an offset to the symbol `btemp`. These variables are no longer used by the OCG compiler. Instead the compiler defines additional auto-style variables for any functions which require additional temporary storage. Assembly code that used temporary variables may need to be revised. Note that these `btemp` variables are saved and restored by the interrupt context switching code in the PICC STD compiler as they are common locations used by all functions. As the temporary auto-style variables used in the OCG compiler are function-specific, they need not, and are not, saved by the interrupt context switching code.

In additional, these `btemp` variables were used in the PICC STD compiler as the parameter and return value locations for some assembly library routines. (These are typically implicitly-called library routines to perform floating-point operations etc.) All assembly library routines now follow the C-style call/return convention. This means that there is only one convention for parameters and return values for all source and library code. Any C-callable assembly routines should be updated to use this same convention.

Assembly code The HI-TECH C Compiler for PIC10/12/16 MCUs is able to extract a much larger amount of information from the C source files in your project. As assembly code (either in-line with C code, or as separate assembly modules) is not processed by the code generator, the use of assembly code can defeat many of the features of the OCG compiler. Now, more than ever, it is highly desirable to write as much of the program in C code rather than assembly to maximize the benefits of the new code generation technology. However, new features have been added to ensure that assembly code integrates seamlessly into C code.

First, you may use variables defined in C code in assembly modules. If you do not use these symbols from C code, the compiler will automatically mark these C variables as `volatile` to prevent them from being removed.

Second, if any psects created in assembly source files are defined as absolute (using the `abs` and `ovlrd`

psect flags) the memory requirements of these psects are noted and passed to the code generator to ensure that this memory is not used by the code generator.